CCP
CONDITIONAL COMMAND PROCESSOR
REFERENCE MANUAL

C. A. Grant

Document No. R-29
July 14, 1967
Contract SD-185
Office of Secretary of Defense
Advanced Research Projects Agency
Washington 25, D. C.

TABLE OF CONTENTS

1.0 Introduction 1-1
2.0 Basic Syntactic Components 2-1
2.1 Numbers 2-1
2.2 Names 2-1
2.3 Variables 2-1
2.4 Dummy Argument Names 2-1
2.5 Dummy Argument References 2-2
2.6 String References 2-2
2.7 Integer References 2-2
2.8 Statements 2-3
2.9 Labels 2-3
3.0 CCP Statements 3-1
3.1 Assignment Statements 3-1
3.2 Internally Coded Working Functions 3-2
3.3 Internally Coded Predicate Functions 3-7
3.4 Character-send Statements 3-10
4.0 Error Handling 4-1
4.1 Compile Time Errors 4-1
4.2 Runtime Errors 4-1
5.0 Running A Program 5-1
6.0 Examples 6-1

1.0 Introduction

In its simplest form, CCP may be used to retrieve characters from a user supplied file and send them to a "pseudo teletype." The pseudo teletype will react to these characters the same way a teletype connected to the system would react in all cases. CCP, then, is a pseudo-typist capable of sending characters to the pseudo teletype.

It is possible to save the output generated by the pseudo teletype and examine it with internally coded CCP functions. A CCP program, therefore, can cause a user program to execute and then examine the resulting output. Conditional statements in CCP will allow appropriate action to take place based on the output.

CCP may be viewed as a macro processor in that arguments can be supplied when a CCP program is to be run. These arguments may be referred to within the program.

CCP takes the form of an algol-like language, including recursive, user-defined functions with substitutable arguments. Additional features are two character-send functions and several other internally coded functions.

2.0 Basic Syntactic Components

This chapter builds the small vocabulary necessary for a full description of CCP.

2.1 Numbers

Only integer numbers are accepted by CCP.

2.2 Names

Names are composed of letters, numbers and blanks. Names may be of any length, but only the first and last four non-blank characters serve to recognize the name. At least one character of a name must be a letter. A name may not refer to more than one syntactic object (e.g., XYZ may not be a label and a variable name).

2.3 Variables

Variables may at any given moment be in one of three states: undefined, string-valued or integer-valued. All variables are initially undefined. Variables may freely change state.

2.4 Dummy argument names

A dummy argument name is a name as defined above, preceded by a ($) dollar sign. So

$ABC
$1B5
$LONG DUMMY NAME

are acceptable dummy argument names.

2.5 Dummy argument references

The use of dummy argument references will be explained below. Syntactically, a dummy argument reference consists of a dummy argument name followed by a parenthesized expression of any complexity involving integers and integer-valued variables. Therefore,

```
$ABC(5)
$1B5(2↑(7-X/3))
```

are syntactically correct dummy argument references.

2.6 String references

A string reference is either a quoted string, a string-valued variable reference, or a string-valued dummy argument reference. For example:

```
'ABC'
'CARRIAGE RETURNS AND LINE
FEEDS AND CONTROL CHARACTERS
MAY BE IN A STRING'
XYZ
$ABC(7)
```

are string references (if XYZ and $ABC(7) are string-valued).

There is a special string-valued variable with the name QUOTE which has the value (') quote-mark. This variable differs from other variables only in the respect that it is initially defined.

There is one other type of string, and this is the "non-string." This is equivalent to a null string but has special meaning, as will be explained later.

2.7 Integer references

An integer reference is either an expression of any complexity involving integers and integer-valued variables, or a dummy

argument reference with an integer value. A dummy argument reference may never appear in an arithmetic expression.

2.8 Statements

There are four types of statements in CCP. They are:

a) Assignment statements
b) Internally coded working functions
c) Internally coded predicate functions
d) Character-send mode statements.

Each type of statement will be fully explained in the next chapter. Any number of statements (or fractions thereof) may appear on one line. Carriage returns, line feeds, and blanks are, except within a quoted string, completely disregarded.

2.9 Labels

Labels are identified by names. A statement (or a labeled statement) may be labeled by preceeding the statement by a label name followed by a (:) colon. A label name may not be used more than once to label a statement.

```
LABEL:
DOUBLY: LABELED:
```

3.0 CCP Statements

3.1 Assignment statements

There are two categories of assignment statements:

a) <Variable> = <String reference>,

b) <Variable> = <Integer reference>,

Note that each type of assignment statement is terminated with a comma.

It is permitted that variables change from string-valued to integer-valued freely. Examples of assignment statements are:

```
A = 3, B = A,
A = 'LOVE',
XYZ = -A↑(B-7/A),
A = $DUMMY(1),
B = $DUMMY(2),
```

3.2 Internally coded working functions

JUMP(<label>)

Execution of this function causes an unconditional transfer of control to the indicated label.

SJUMP(<label>)

Transfer to the label takes place only if the current predicate value is success (see next section). Otherwise, the flow of control passes to the next statement in the program.

FJUMP(<label>)

Transfer is effected only if the current predicate value is failure.

FUNCTION(<label>, <dummy argument name>, <list of local variables>)

This statement causes a function to be associated with the label. References to arguments given to the function when called will be made with the indicated dummy argument name. The list of variables will be considered local to the function. There may be only one FUNCTION statement for a given label. Two or more different functions may use the same dummy argument name. Examples:

```
FUNCTION (ABC,$DUMMY)
FUNCTION (XYZ, $DUM, TEMP1, TEMP2)
```

CALL(<label>, <list of arguments>)

This statement will call the function associated with the label. If a call to ABC (defined above) is executed:

```
CALL(ABC, 100, 'PEACE')
```

then $DUMMY(1) will be integer-valued with the value 100 and $DUMMY(2) will be string-valued with the value 'PEACE'. The zero-th reference is always integer-valued with the

number of arguments supplied as its value. So in this instance $DUMMY(0) is equal to two. References to $DUMMY(N) where N is less than zero or greater than two will be errors. Labels must be passed as arguments enclosed in (") double-quote marks. So

```
        CALL(ABC, "LABEL")
              :
              :
        ABC:  JUMP($DUMMY(1))
```

causes a transfer to the statement labeled with LABEL. Arguments provided in a CALL statement must be string references, expressions of any complexity, labels or dummy argument references. So

```
CALL(ABC,7,X↑5-Y/Z, 'MOTHER', $XYZ(7), "LABEL")
```

is syntactically correct.

To return from a called function, there are three ways:

JUMP(RETURN) - causes a return to the next lower level, reinstating the value of predicacy that existed when the function was called.

JUMP(SRETURN) - causes a return, and changes the value of predicacy at the lower level to success.

JUMP(FRETURN) - causes a return and changes the value of predicacy at the lower level to failure.

SCALL(<label>, <list of arguments>)

The CALL is executed only if the current value of predicacy is success.

FCALL(<list of arguments>)

The CALL is executed only if the current value of predicacy is failure.

CONCAT(<variable>, <string reference>, <string reference>)

Execution of this statement causes the value of the variable to become the string obtained by concatenating the two string references.

CNSTNU(<variable>, <string reference>)

This converts the first numeric string of characters in the string reference to a decimal integer and sets the value of the variable to this integer. A (+) plus sign or (-) minus sign will be considered numeric characters.

e.g. CNSTNU(X, 'ABC123ABC') cause X to become 123
CNSTNU(X, '10') cause X to become 10
CNSTNU(X, 'A-A5') cause X to become 0

CNNUST(<variable>, <integer reference>)

This function causes the integer to become converted into a string of digits, signed only if negative, and stored as the value of the variable.

ERCOMP(<label>)

Execution of this function causes a transfer to the label only if during compile-time errors were detected in the program.

ERJUMP(<label>)

This function saves the label and will cause transfer to the label in the event that a run-time error occurs. If a second run-time error occurs before another ERJUMP statement is executed, execution halts.

COMMENT(<string reference>)

This function causes the string to be printed on the teletype when this function is executed.

TTYON(<integer reference>)

This function determines whether or not output of the pseudo teletype is to be printed on the controlling (user) teletype. Initially, the output is not printed. To cause the output to be printed, execute TTYON with the value of the integer reference non-negative. To turn

the teletype off, execute TTYON with a negative argument. Note that TTYON and OUTFILE are completely independent of each other.

OUTFILE(<string reference>)

This function opens the file specified by the string reference and causes all output of the pseudo teletype to be diverted to the file. The file is opened at its beginning.

REOPEN(<string reference>)

Same as above except file is opened at its end (i.e., output from the pseudo teletype is added to the contents of this file.)

SETFLAG(<string reference>)

The string referenced is written on the current outfile between two control ([) left-bracket characters. This flag may be used by the predicate functions MATCH and GSTRING, to be described below.

ERSFLAG(<string reference>)

This function erases all flags written on the file indicated by the string reference.

TIME(<integer reference>, <label>)

This function specifies that a transfer to the indicated label is effected if and only if the next character-send statement does not terminate within N seconds, where N is the value of this integer reference.

INTERACT(<mode character>)

Read about character-send statements before trying to understand this function. This function may be used to

allow interaction to take place between the user (via the teletype) and the pseudo teletype. The mode character (@ or >) determines whether @-mode or >-mode is desired. All characters are sent literally except that control (←) left arrow causes a rubout to be sent, control ([) left-bracket causes termination of the mode, and rubout terminates the CCP program. When INTERACT is executed, several bells will ring to let the user know.

3.3 Internally Coded Predicate Functions

During execution of a CCP program, there is a state of being, known as predicacy, which exists with one of two values: success or failure. Initially (when a program starts) and each time a function is called, predicacy is automatically set to success. Thereafter, the only ways the value of predicacy may be changed are by executing one of the following predicate functions, or by returning from a called function via SRETURN or FRETURN. Predicacy may be tested for by any of the previously explained functions SCALL, FCALL, SJUMP, and FJUMP.

NULL(<string reference>)

Sets predicacy to success if the string reference is the null string, else sets predicacy to failure.

EQUAL(<integer reference>, <integer reference>)

Success if the two integers are the same value, else failure.

GRTR(<integer reference>, <integer reference>)

Success if the first integer is greater than the second, else failure.

STREQL(<string reference>, <string reference>)

Success if the two strings are exactly the same, else failure.

MATCH(<string ref1>, <string ref2>, <string ref3>, <string ref4>)

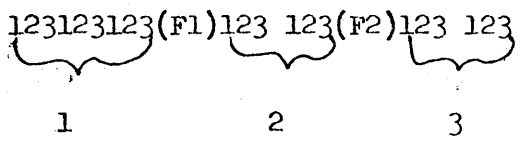
This function executes a search on the file specified by string reference 1. The string referenced by string reference 2 is the string of characters searched for. MATCH changes the value of predicacy to success if the string is found, else changes the value to failure. The

bounds of the search are indicated with string references 3 and 4 in the following complicated manner.

If string reference 3 is a flag by SETFLAG, the search commences after the first occurrence of this flag. If string reference 3 is the (") null string, the search will begin after the point where the search was most recently discontinued. If the () non-string is specified, then the search begins at the beginning of the file (see example below for clarification).

String reference 4 determines where the search will terminate. If string reference 4 is a flag set by SETFLAG, then the search will terminate at the first occurrence of this flag after the search has started. If string reference 4 is the () non-string, or if the flag is not encountered in the file during the search, then the search will terminate at the end of the file.

Assume the file /X/ has the following characters, with flags (F1) and (F2):



Then the following program will count the occurrences of the string '12' in regions 1 and 3:

```
            COUNT = 0,
            MATCH('/X/', '12',,'F1')        FJUMP(REGION3)
LOOP1:      COUNT = COUNT + 1,
            MATCH('/X/', '12', '', 'F1')    SJUMP(LOOP1)
REGION3:    MATCH('/X/', '12', 'F2')        FJUMP(EXIT)
LOOP3:      COUNT = COUNT + 1,
            MATCH('/X/', '12', '')          SJUMP(LOOP3)
EXIT:
```

GSTRING(<string ref 1>, <variable>, <string ref 2>, <string ref 3>,
<string ref 4>, <string ref 5>)

This function executes a search on the file specified by string reference 1. It searches for a string of characters preceded by the string referenced by string reference 2 and followed by the string referenced by string reference 3. If such a string is found, then the value of the variable is set to this string, and the value of predicacy is set to success. If not found, the variable is set to the null string and the value of predicacy is set to failure. String references 4 and 5 specify the bounds of the search in the same way the bounds are set in the function MATCH.

Consider the file /Y/ whose contents, with flags (F3) and (F4) are as follows:
ABCDCD7F(F3)/1/2/3/4/(F4)1/1/1/

The statement

GSTRING('/Y/', X, 'CD', '7')
succeeds with X = 'CD7'.

The following program calls the user function EXAMINE for each number found between (/) slashes on the file:

```
        GSTRING('/Y/',X,'/','/')        FJUMP(EXIT)
LOOP:   CALL(EXAMINE,X)
        GSTRING('/Y/',X,'/','/',")      SJUMP(LOOP)
EXIT:
```

Carefully notice that this program will call EXAMINE for all 7 cases. The flag (F4) will be ignored completely.

3.4 Character-send statements

There are two modes for sending characters to the pseudo-teletype. @-mode insures that the Time-Sharing executive is the listening program by sending several rubouts. >-mode does not disturb the pseudo teletype before sending characters. For example, the CCP statement

@CAL.!

first causes several rubouts to be sent, then a C, an A, a L, and finally a (.) period. The exclamation point indicates the termination of the statement. Character-send statements may also be terminated with a (%) percent sign, and the distinction will be described below. If the next CCP character-send statement is

>SET A = 1!

Then the characters

SET A = 1

would be sent, and CAL would still be listening. The pseudo teletype is initially set in BEGINNER mode (see document $-22).

It is possible for the value of a string-valued dummy argument reference to be sent in character-send mode, and this is indicated by including the reference in the statement:

COPY FILE $DUMMY(3) TO $DUMMY(4).%

if $DUMMY(3) is 'XYZ' and $DUMMY(4) is '/XYZ' then the characters sent will be several rubouts and then:

COPY FILE XYZ TO /XYZ.

To cause the string value of a variable to be sent, the variable name must be preceded by a ($) dollar sign and followed by a (.) period: If X has the value '/FILE/', then

>GO TO $X..!

causes the string

GO TO /FILE/.

to be sent.

If a control (←) left arrow is found in the statement, then a rubout will be sent to the pseudo teletype instead of that character. If a control ([) left-bracket is found in the statement, then the statement will immediately terminate (as if the (!) exclamation point had appeared at that point.) Clearly the value of this convention is seen only where variable and dummy argument references are involved. Carriage returns and line feeds are not normally sent.

```
@QED.
$X..
1,5
YES
  c c c
←←←
$Y
COPY FILE /A/ TO /B/.
```

If X has the value '/A/' and Y has the value 'AB]^{c}CD', then the characters sent will be

QED./A/.1,5.YES rubout rubout AB

In the sending of variable and dummy argument references, all characters are sent literally except control (←) left arrow and control ([) left bracket. i.e., if X is '$Y.' then

```
>$X.!
```

causes the characters

$Y.

to be sent.

If it is desired that the characters $, !, %, $\leftarrow^{c}$, $[^{c}$, cr, lf be sent without the above-mentioned conventions, then they must be preceded by a ($) dollar sign. (This does not hold within variable or dummy argument references.) A ($) dollar sign found in any context other than those heretofore described will cause a compile error. The CCP statement

>$:$$:

sends

:$

to the pseudo teletype.

Now the difference between terminating a character-send mode statement with (:) exclamation point or (%) will be explained. When (:) exclamation point is used, the flow of control will not pass to the next statement until the last character has been sent and the pseudo teletype is again waiting for input. Termination with (%) causes completion of the statement as soon as the last character is sent.

4.0 Error Handling

4.1 Compile Time Errors

Syntactic errors will be discovered at compile-time and error messages giving the line number and an explanation will be generated to the teletype. After each discovered syntactic error, CCP will search through the input text for a labeled statement. At this point compilation will continue. Whether or not there were compilation errors can be tested at runtime with the function ERCOMP.

When a name is found to have double use, an error message is generated and the first use remains in effect (e.g., doubly used label, or a name first used as a variable and then as a label).

4.2 Runtime Errors

Runtime errors will likewise result in hopefully elucidative error messages. Possible runtime errors are: finding an out-of-bounds dummy argument reference, using a wrong-type variable or trying to execute a statement which did not compile correctly.

When a runtime error is encountered, a check is made to see if an ERJUMP statement has been executed. If such a statement was executed, and no runtime errors have occurred since, then the stack is reset to level 0 and a transfer to the ERJUMP label is executed. A message is printed on the teletype indicating this activity. Otherwise, execution of the program is terminated.

5.0 Running a Program

CCP programs are best composed in QED. If serious work is being done, it is suggested that free use of the functions ERCOMP and ERJUMP is made.

When calling the subsystem CCP, a list of arguments is requested. Arguments may be supplied in exactly the same manner as in a CALL function. Typing a ()) right parenthesis will terminate the list, and CCP will request the name of the file on which the program lies. A typical encounter with CCP might appear as follows (underlined characters are typed by CCP):

@CCF.

ARGUMENTS: (27, '/FILE', "LABEL1", 2↑5)
INPUT: /CCP.

BAD EXPRESSION AT LINE+2
NAME USED WRONGLY LINE+7
'STEFLAG' IS NOT A CCP FUNCTION AT LINE 12

*** COMPILED WITH 3 ERRORS ***

COMPILE ERROR ENCOUNTERED
LINE 2, LEVEL 0
ERJUMP TRANSFER TO LABEL 1

OUT-OF-BOUNDS DUMMY REFERENCE
LINE 36, LEVEL 7
NO ERJUMP TRANSFER

END

@

The arguments given to CCP at runtime are referred to with a special dummy argument name. In the above example:

$(1) is 27
$(2) is '/FILE' etc.
and
$(0) is 4

6.0 Examples

This program expects a list of files in pairs that are to be copied. When the argument '!' is found, the rest of the arguments are files to be assembled. Checks are made during the copying that no errors occurred.

```
        ERCOMP(EXIT)
        FUNCTION(COPYCHECK,$DUM)
        N = 1,
LOOP1:  STREQL($(N), '!')  SJUMP(ASSEMBLE)
        OUTFILE('/$X')
        @COPY FILE $(N) TO $(N+1).!
        CALL (COPYCHECK)
        N = N+2, JUMP(LOOP1)

ASSEMBLE:  EQUAL($(0),N)  SJUMP(EXIT)
           @ARPAS.$(N+1),$(N+2).!
           N = N+2, JUMP(ASSEMBLE)

COPYCHECK:  MATCH('/$X', '. cr lf lf')  SJUMP(RETURN)
            INTERACT(>)  JUMP(RETURN)

EXIT:
```